注册编译入口
registerRuntimeCompiler(compileToFunction)
(vue/index) 查看最右侧连接点
compileToFunction
(vue/index)
const { code } = compile(template, opts)
通过编译得到 code 代码
compile
(compiler-dom/index)
baseCompile
(compiler-core/compile)
return
generate(ast, options)
(compiler-core/codegen)
解构出来的 code
{ code }
isVoidTag
是否空
isNaiveTag
是否原生
isPreTag
是否'pre'
decodeEntities
decode
isBuiltInComponent
内置过渡组件
getNameSpace
命名空间
getTextNode
特殊文本节点
DOMNodeTransforms
style transition
ignoreSideEffectTags
script style 副作用tag
DOMDirectiveTransforms
cloak html text model on show
parseOptions
nodeTransforms
节点转换
directiveTransforms
指令转换
const render = (__GLOBAL__ ? new Function(code)() : new Function('Vue', code)(runtimeDom))
走前 (Component.render = render)
生成 组件的 render 函数
const ast = isString(template) ? baseParse(template, options) : template
baseParse
(compiler-core/parse) 解析模版转为ast
构建配置context
创建指针并初位置
优化和标记后的 AST
携带 ast.helpers函数,
供生成代码时导入
openBlock,
resolveComponent等
走 baseParse，解析
createRoot(parseChildren(), getSelection())
创建Root，使用parseChildren
解析 模版
(compiler-core/ast)
标记解析Vue语法
transform(ast, options)
(compiler-core/transform) 解析
Vue语法，静态标记等
1. createTransformContext
创建转换器上下文
2. traverseNode(root, context)
递归遍历补充helpers函数
3. hoistStatic
做静态提升
walk
做静态标记
(compiler-core/hoistStatic)
4. createRootCodegen
创建根codegen,
root.codegenNode
helpers: new Map(),辅助函数
components: new Set(),
directives: new Set(),
imports: [],
hoists: [], 提升容器
hoist(exp) {}, 提升函数
cache(exp){}, 缓存函数
context 转换器上下文
directiveTransforms: {bind(), cloak(), html(), model(), on(), model(), show(), text()},
nodeTransforms{transform(Once/Memo/Expression
/SlotOutlet/Element/SlotScopes/Text/Style/Transform),
ignoreSideEffectTags},等
transform 的配置项参数 指令和节点
parseChildren
判断以 '<' 开始, !,/,a~z,?
(compiler-core/parse)
parseInterpolation
以 '{{' 开头的插值
(compiler-core/parse)
parseText
文本
(compiler-core/parse)
parseElement
判断以 '<a~z' 开始的元素
(compiler-core/parse)
parseTag
<div :id=a>,开始和结束tag
(compiler-core/parse)
tag: 'h3'/'span' 等,
tagType:
EIEMENT,SLOT,TEMPLATE,COMPONENT,
props: [{arg: 'id',exp: 'a',name: 'bind'}]
标签名称，类型，props是attributes，等
parseAttributes
解析标签上的属性
(compiler-core/parse)
attr = parseAttribute()
v-xx bind on slot
(compiler-core/parse)
1. 创建codegen上下文
createCodegenContext
2. genFunctionPreamble() 'with 模式'
1. 'const _Vue = Vue'
2. 'const {xx: _xx} = _Vue'
3. genHoists()'静态提升变量'
4. '新行 return '
3. 'function render(_ctx, _cache) {'
4. with (_ctx) {
5. 插进，'const {xx: _xx} = _Vue'
6. 换行，新行，genAssets()解析
(componnets, directives, v2 filters)
7. 创建变量，'let _temp0 , _temp1'
8. 'return genNode(ast.codegenNode, context)'
9. 退缩进，补 '}'
generate 生成代码过程
genFunctionPreamble
(compiler-core/codegen)
genHoists(ast.hoists, context)
(compiler-core/codegen)
genAssets 生成资源
(compiler-core/codegen)
判断type
const _directive_model = _resloveDirective("model")
const _component_Comps = _resolveComponent("Comps")
生成的代码
genNode(ast.codegenNode, context)
(compiler-core/codegen) 递归遍历
通过 ast.helpers的辅助函数导入
api，生成需要的 code
const _Vue = Vue
return function render(_ctx, _cache) {
with (_ctx) {
通过 ast.helpers.keys() 提供转换
const { renderList: _renderList, Fragment: _Fragment, openBlock: _openBlock, createElementBlock: _createElementBlock, toDisplayString: _toDisplayString } = _Vue
return (_openBlock(), _createElementBlock("div", null, [
(_openBlock(true), _createElementBlock(_Fragment, null, _renderList(this.msg, (i) => {
return (_openBlock(), _createElementBlock("h3", { key: i.id }, _toDisplayString(i.id) + " -- " + _toDisplayString(i.title), 1 /* TEXT */))
}), 128 /* KEYED_FRAGMENT */)
]))
}
}
createCodegenContext
(compiler-core/codegen)
helper() 辅助函数,
push(code) { 不断生成code
context.code += code},
code: '', 存放生成的code
map: context.map.toJSON()
codegen上下文
patch
(runtime-core/renderer)
首次挂载 响应式数据变更时更新组件 递归执行 patch
Text
processText()
文本
Comment
processCommentNode()
Static
n1--null
mountStaticNode()
__DEV__
patchStaticNode()
Fragment
processFragment()
ELEMENT
processElement()
html 标签
COMPONENT
processComponent()
组件
TELEPORT
type.process()
SUSPENSE
type.process()
1. hostInsert
2. hostSetText
1. hostInsert
2. hostCreateComment
1. hostInsertStaticConent
1. hostInsertStaticConent
2. hostNextSibing
3. removeStaticNode
mountChildren()
挂载子节点
dynamicChildren
diff
patchBlockChildren()
动态节点 block树
patchChildren()
patch 子节点 Diff
1. hostInsert
mountElement()
挂载元素
patchElement()
patch 元素
mountComponent()
组件首次渲染
updateComponent()
组件更新渲染
type: rootComponent, 组件
props: rootProps,
scopeId: currentScopeId,
appContext: context,
shapeFlag: 组件类型,
patchFlag: patch类型,
dynamicProps: 动态props,
dynamicChildren: 动态子节点,
render: null,
renderCache: [],
root: null,
next: null,
proxy: null,
provides: 寻找parent,
scope: new EffectScope(true)
vnode包含的东西？
return vnode
baseCreateVNode
(runtime-core/vnode)
createVNode
(runtime-core/vnode)
返回 vnode
unmount
卸载
卸载老的vnode
vnode==null
render
(runtime-core/renderer)
flushPreFlushCbs()
flushPostFlushCbs()
render来源
const vnode = createVNode(rootComponent, rootProps)
vnode 传入render
render(vnode, rootContainer)
SPA
hydrate(vnode, rootContainer)
SSR
devtoolsInitApp(app, version)
devtool
mount()
const proxy = mount(container)
app
version
config
use()
mixin()
component(name, component)
directive(name, directive)
unmount()
provide(key, value)
pkg.version
context.config
const installedPlugins = new Set()
context.components[name] = component
context.directives[name] = directive
render(null, app._container)
devtoolsUnmountApp(app)
context.provides[key as string | symbol] = value
_uid: uid++
_component: rootComponent
_props: rootProps
_container: rootContainer
_context: context
_instance 组件实例
app 包含的东西？

{app, config: {}, mixins: [], components: {}, provides: {}}
app._instance = vnode.component
编写入口
const app = createApp(App)
...args
const app = ensureRenderer().createApp(...args)
createApp
(runtime-dom/index)
ensureRenderer
(runtime-dom/index)
rendererOptions
创建渲染器
createRenderer
(runtime-core/renderer)
options
baseCreateRenderer
(runtime-core/renderer)
return
hydrate
(runtime-core/renderer)
render传入
ensureRenderer 调用返回 createApp
createApp是createAppAPI 调用的返回
返回app
解构传入的options
renderer = createRenderer(rendererOptions)
参数
1. nodeOps
2. patchProp
insert() 插入元素
remove() 移除元素
createElement() 创建元素
createText() 创建文本
createComment() 创建注释
setText() 设置文本
setText() 设置文本
createElementText() 设置元素内容
insertStaticContent() 设置静态内容
setScopeId() 设置ScopeId
querySelector() 查询选择器
patchClass() 类名
patchStyle() 样式
patchEvent() 事件
patchDOMProp() prop
patchAttr() attr
const {
insert: hostInsert,
remove: hostRemove,
patchProp: hostPatchProp,
createElement: hostCreateElement,
createText: hostCreateText,
createComment: hostCreateComment,
setText: hostSetText,
setElementText: hostSetElementText,
parentNode: hostParentNode,
nextSibling: hostNextSibling,
setScopeId: hostSetScopeId = NOOP,
cloneNode: hostCloneNode,
insertStaticContent: hostInsertStaticContent
} = options
const proxy = app.mount('#app')
app.mount: (containerOrSelector) => {}
(runtime-dom/index)
return getExposeProxy(vnode.component!) || vnode.component!.proxy
createAppAPI
(runtime-core/apiCreateApp)
createApp: createAppAPI(render, hydrate)
(runtime-core/renderer)
render
createAppAPI
(runtime-core/apiCreateApp)
createApp(rootComponent)
(runtime-core/apiCreateApp)
...args
{
setup() {},
render() {}
}
最快路径，有patchFlag
patchFlag > 0
patchKeyedChildren()
patchUnkeyedChildren()
没有 key
1. 获取最小长度 length
2. 遍历 patch
3. 判断 如果 oldLength > newLength
4. unmountChildren(c1) 移除旧的
5. 否则 挂载新的 mountChildren(c2)
子元素情况不同
新子元素是 文本
老子元素是 数组
老子元素是 文本
老子元素是 空
1. unmountChildren
2. 创建文本
设置文本
设置文本
新子元素是 空
老子元素是 数组
老子元素是 文本
老子元素是 空
unmountChildren
删除文本
不需要操作
新子元素是 数组
老子元素是 数组
老子元素是 文本
老子元素是 空
diff 操作
patchKeyedChildren()
mountChildren
mountChildren
1. 相同的头部 patch
2. 相同的尾部 patch
3. 公共序列 和 挂载
4. 公共序列 和 卸载
5. 未知序列
initProps(instance, props)
初始化 props
(runtime-core/componentProps)
initSlots(instance, children)
初始化 slots
(runtime-core/componentSlots)
emit
emit 函数
(runtime-core/componentEmits)
instance.root
instance.emit
组件实例
const instance = {}
setupStatefulComponent
有状态组件的setup函数
(runtime-core/component)
_组件实例 instance 包含的东西？
createSetupContext(instance)
创建setupContext，setup 函数第二个参数
(runtime-core/component)
handleSetupResult(instance, setupResult)
处理setup 的返回值
(runtime-core/component)
isFunction
直接将setupResult赋值给render
(直接返回渲染函数的写法)
isObject
instance.setupState = proxyRefs(setupResult)
处理后的结果储存在 setupState 中
正常return {}
执行完后执行下面
finishComponentSetup(instance)
完成组件setup操作, 最后都会执行
(runtime-core/component)
const Component = instance.type
compile && !Component.render
template 存在
Component.render = compile(template, finalCompilerOptions)
handleSetupResult(instance, setupResult)
处理setup 的返回值
(runtime-core/component)
compile = compileToFunction
将编译器赋值，使compile为 true
(reactivity/ref)
registerRuntimeCompiler
注册运行时编译器
(runtime-core/component)
template 到编译器转 render 函数连接点
instance.render = Component.render
获取到 render 函数
(runtime-core/component)
挂载
1. beforeMount
2. const subTree = (instance.subTree = renderComponentRoot(instance))
3. mounted
先执行 beforeMount，subTree 为当前组件的子树，渲染完成后执行 mounted
! instance.isMounted
if()
instance.isMounted
更新
1. updateComponentPreRender(instance, next, optimized)
2. beforeUpdate
3. const nextTree = renderComponentRoot(instance)
4. const prevTree = instance.subTree
5. instance.subTree = nextTree
6. patch(prevTree, nextTree)
7. next.el = nextTree.el
8. updated
执行前先使用 updateComponentPreRender 更新 props 和 slots 等，然后开始执行 beforeUpdate，获取新旧子树，进行patch，渲染完成后执行 updated
updateProps(instance, nextVNode.props, prevProps, optimized)
updateSlots(instance, nextVNode.children, optimized)
(runtime-core/renderer)
renderComponentRoot()
(runtime-core/componentRenderUtils)
componentUpdateFn
组件更新函数
const effect = new ReactiveEffect(componentUpdateFn, () => queueJob(update), instance.scope)
响应式副作用 effect
(runtime-core/renderer)
const update = instance.update = () => effect.run()
更新函数 update
(runtime-core/renderer)
setupRenderEffect()
监听副作用函数
(runtime-core/renderer)
响应式Effect
return {
get attrs() {
return attrs || (attrs = createAttrsProxy(instance))
},
slots: instance.slots,
emit: instance.emit,
expose
}
createAttrsProxy(instance)
创建 attrs 的代理，触发 track
(runtime-core/component)
get(target, key: string) {
track(
instance,
TrackOpTypes.GET, '$attrs')
return target[key]
}
new Proxy(objectWithRefs, shallowUnwrapHandlers)
shallowUnwrapHandlers
使用 unref()
直接返回
isReactive
proxyRefs
在模版中去 .value 的处理
(reactivity/ref)
setupStatefulComponent
有状态组件的setup函数
(runtime-core/component)
setupComponent(instance)
执行 setup 函数
(runtime-core/component)
响应式系统 ReactiveEffect
响应式系统和render的连接点
ReactiveEffect
(reactivity/effect)
proxy代理
target
reactive
ShallowReactive
readonly
ShallowReadonly
ref
const proxy = new Proxy(target, baseHandlers)
createReactiveObject
(reactivity/reactive)
mutableHandlers
(reactivity/baseHandlers)
get: createGetter()
set: createSetter()
deleteProperty
has
ownKeys
const result = Reflect.has(target, key)
return Reflect.ownKeys(target)
const res = Reflect.get(target, key, receiver)
const result = Reflect.deleteProperty(target, key)
const result = Reflect.set(target, key, value, receiver)
get
set
track
trigger
track
trigger
track
(reactivity/effect)
trigger
(reactivity/effect)
响应式数据结构
执行 所有的effect函数
track(target, TrackOpTypes.GET, key)
没有key 添加: trigger(target, TriggerOpTypes.ADD, key, value)
设置值: trigger(target, TriggerOpTypes.SET, key, value, oldValue)
targetMap = new WeakMap(target, depsMap)
depsMap = new Map(key, dep)
dep = createDep()
const dep = new Set()
for (const dep of deps){effects.push(...dep)}
triggerEffects(createDep(effects))
1. scheduler
2. run
triggerEffect
判断如果有scheduler执行 effect.scheduler(),
否则执行 effect.run()